EECS 280 – Lecture 13

Memory Models and Dynamic Memory

Recall: Compound Object Lifetimes

- Constructors are called whenever a class object is created for the first time.

```
Bird(const string &name_in)
  : name(name_in) {
  cout << "Bird ctor: " << name << endl;
}
```

- Destructors are called whenever a class object's lifetime ends (depends on storage duration).
  - e.g. For local variables, when they go out of scope.

```
~Bird() {
  cout << "Bird dtor: " << name << endl;
}
```

① constructor 是何时被 call

② destructor 是何时被 call

Example: Local Object Lifetimes

```
Bird(int id_in)
  : ID(id_in) {
  cout << "Bird ctor: " << ID << endl;
}

~Bird() {
  cout << "Bird dtor: " << ID << endl;
}

Bird b_global(0);

int main() {
  Bird b1(1);
  for (int i = 0; i < 3; ++i) {
    Bird b2(2);
    b2.talk();
  }
  b1.talk();
  if (100 < 2) {
    Bird b3(3);
    b3.talk();
  }
  else {
    Bird *ptrToB1 = &b1;
    ptrToB1->talk();
  }
}
```

there will not have a constructor in the output, since we only creat a pointer points to a class, but not creat a new class

Output

```
Bird ctor 0
Bird ctor 1
Bird ctor 2
Bird ctor 2
Bird dtor 2
Bird ctor 2
Bird dtor 2
Bird ctor 2
Bird dtor 2
Bird dtor 1
Bird dtor 0
```

the class created first will be deleted at least, since the new class may depend on the old class

5/25/2022

① global variable 的 ctor 和 dtor

② local variable 的 ctor 和 dtor

③ 在 for loop 中的 local variable 的 ctor 和 dtor

Recall: C++ Memory Model

- An **object** is a piece of data in memory.
- An object lives at an **address** in memory.
- You can use an object during its **lifetime**.
- Lifetimes are managed according to **storage duration**. Three options in C++:

Managed by the compiler.
- ***Static***
  Lives for the whole program.
- ***Automatic (Local)***
  Lives during the execution of its local block.

Managed by you
- ***Dynamic***
  You control the lifetime!

① object 在 memory 中的本质是什么.

② object 何种形式存在于 memory. 何时能使用 object. ③ Storage duration 3类.

Automatic Storage (Local Variables)

Managed by the compiler.

- A local variable lives inside a **block**.
- A block is a set of curly braces.
- Usually a function or loop body, but you can also make just a plain block.
- Initialized when the declaration is run.
- Dies when the block is finished.
- Scope *usually* keeps us from using a dead local.

① 对于 local variable 来说, 什么意味着开始, 什么意味着结束.

Memory Address Space

- The operating system allows each running program to use a particular range of addresses.
- This is called the program's **address space**.
- It is divided into several segments...

Memory Address Space

Automatic Storage
Stack frames are added and removed as functions are called.

Static Storage
Objects that can be used throughout the whole program.

```
Address MAX
Stack (grows down)
Free space
Globals (Fixed size)
Text (The program)
Address 0
```

Machine code for your program.

5/25/2022

Memory Address Space

Automatic Storage
Stack frames are added and removed as functions are called.

Dynamic Storage
Individual objects created on demand.

Static Storage
Objects that can be used throughout the whole program.

```
Address MAX
Stack (grows down)
Free space
Heap (grows up)
Globals (Fixed size)
Text (The program)
Address 0
```

Machine code for your program.

5/25/2022

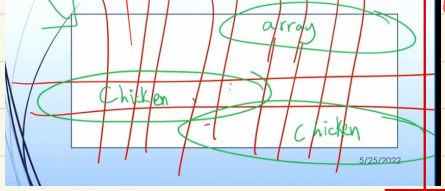
① memory 中的5个空间.

② Bird_Factory 在该 Model 下的工作原理.

③ 不同 variable type 存储位置.

What is the Heap?

- The heap is a separate region of memory used for **dynamic storage**.
- Each object on the heap has its own lifetime. (No "frames" like the stack.)



5/25/2022

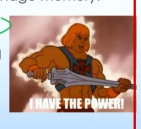
① heap 中空间分配原理.

② heap 中 object lifetime 自控.

Dynamic Storage

Managed by the programmer!

- We just saw one example of when we might not want the compiler to automatically destroy objects in memory.
- **Dynamic memory** lets *us* manage memory!
  - Objects are created on the heap by a new expression.
  - Those objects are destroyed by a delete expression.
  - You put these in the code yourself! You decide!



* The malloc and free functions also work with the heap, but they aren't generally used in C++ style programming.

5/25/2022

① 如何在 heap 中创造并删除一个 object.

The new Operator

- Use new to create a dynamic object.

```
int *ptr = new int(3);
```

- Here's what happens:
  - Space for an int is created on the **heap**.
  - The int is initialized with value 3.
  - The new expression *evaluates* to the address of the object.

The lifetime of the pointer ptr is not controlled by human, instand it is on the stack memory space.

The dynamic memory don't have name

5/25/2022

① "new" 表示返回的是 heap 中的一个地址

② heap 中开辟的空间根据 variable type 来定

③ variable 没有名字且注意 "()" "int a = 3" vs "int(3)"

```
the compiler will give us an error, since we only
creat a double variable type in the dinemic memory
space, but we don't record where we stall this
variable!
new double(3.5);
```

5/25/2022

① 在 heap 中开辟空间和地址数据后 → 一定要接收

The Factory Pattern

- A factory function creates and returns objects to be used elsewhere.
- Dynamic memory allows the object to live beyond the scope of the factory function.

```
// REQUIRES: Color must be blue or black
// EFFECTS:  Creates a bird with the given color
//           and returns a pointer to it.
Bird * Bird_factory(const string &color, const string &name) {
  if (color == "blue") {
    return new BlueBird(name);
  }
  else if (color == "black") {
    return new Raven(name);
  }
  else {
    assert(false);
  }
}
```

```
int main() {
  string color;
  cin >> color; // User enters "black"
  Bird *bird = Bird_factory(color, "Poe");
  bird->talk(); // Prints "nevermore!"
  // One more line missing here...
}
```

delete bird;

5/25/2022

① Bird_Factory 的实现方式

② "main()" 中从接收 address 再到 "delete"

Keeping Track of Dynamic Objects

- For a **local object** (automatic storage)...
  - The lifetime of the object is tied to the scope of the variable's declaration!
  - When the variable goes out of scope, the object dies and can't be used.
- But a dynamic object isn't created from a declaration that has any particular scope!
- We keep track of it using a pointer that holds its address.

```
int *ptr = new int(3);
```

① local object 和 dynamic object 之间的区别

② 如何用 pointer track the address of dynamic object

The delete Operator

- Use delete to destroy a dynamic object.
- The operand is the address of the dynamic object (i.e. a pointer to it).

```
int *ptr = new int(3);
delete ptr;
```

actually, it should called delete through the ptr: since we need to find the address of the variable stored in the dnemic memory and then delete that dynemic variable.

- Here's what happens:
  - Follow the pointer to a dynamic object.
  - Destroy whatever object was there.

① delete 的工作原理是什么.

Using delete

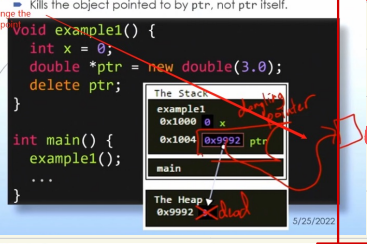
the pointer itself is not dead, it just points to some address which are already dead.

But we change change the pointer in order to point to other address

- delete follows the pointer and kills the object found.
- Kills the object pointed to by ptr, not ptr itself.

```
void example1() {
  int x = 0;
  double *ptr = new double(3.0);
  delete ptr;
}

int main() {
  example1();
  ...
}
```



5/25/2022

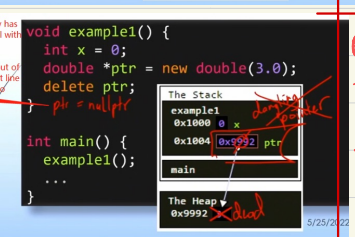
① delete 是对对 object 本身而不是 pointer

② dangling pointer 的使用注意.

here, we actually has two ways to deal with "ptr":
① do nothing
since the ptr is out of scope in the next line
② let ptr points to "nullptr"

```
void example1() {
  int x = 0;
  double *ptr = new double(3.0);
  delete ptr;
}

int main() {
  example1();
  ...
}
```



5/25/2022

① 对于 dangling pointer 的两种可行的处理方式

Simple Pattern for new/delete

```
void rotate_left(Image* img) {

  int width = Image_width(img);
  int height = Image_height(img);

  // auxiliary image to temporarily store rotated image
  Image *aux = new Image;
  Image_init(aux, height, width); // width and height switched

  // iterate through pixels and place each where it goes in temp
  for (int r = 0; r < height; ++r) {
    for (int c = 0; c < width; ++c) {
      Image_set_pixel(aux, width - 1 - c, r, Image_get_pixel(img, r, c));
    }
  }

  // copy data back into original
  *img = *aux;
  delete aux;
}
```

you need a bigger space for img, if you put img into stack space, the programe will crash

5/25/2022

① 一个对于 new 和 delete 使用的例子.

The Factory Pattern

- A factory function creates and returns objects to be used elsewhere.
- Dynamic memory allows the object to live beyond the scope of the factory function.

```
// REQUIRES: Color must be blue or black
// EFFECTS:  Creates a bird with the given color
//           and returns a pointer to it.
Bird * Bird_factory(const string &color, const string &name) {
  if (color == "blue") {
    return new BlueBird(name);
  }
  else if (color == "black") {
    return new Raven(name);
  }
  else {
    assert(false);
  }
}
```

```
int main() {
  string color;
  cin >> color; // User enters "black"
  Bird *bird = Bird_factory(color, "Poe");
  bird->talk(); // Prints "nevermore!"
  delete bird;
}
```

more code

Remember to delete when you're done with it!

5/25/2022

① Bird_Factory 的工作原理.

② 结尾要 delete.

Reprise: Class-Type Objects

- When a class-type object is created, its **constructor** runs.
  - For a **local** object, when its declaration runs.
  - For a **dynamic** object, when it is created with new.
- When a class-type object dies, its **destructor** runs.
  - For a **local** object, when it goes out of scope.
  - For a **dynamic** object, when it is destroyed with delete.
- The lifetimes of the members of a class-type object are tied to the lifetime of the whole object itself.
  - After its body executes, the destructor of a class-type object automatically runs the destructors of its members that are of class type.

The lifetimes of array elements are tied to the lifetime of the array as a whole.

5/25/2022

① local object ctor 和 dtor 的使用时机.

② dynamic object ctor 和 dtor 的使用时机.

③ class object 中的 member object 和整体一起 d-tor.

Exercise: Whac-a-Mole

heap

http://bit.ly/3eiYbnJ

Question
At each line, how many Mole objects are alive?

```
class Mole {
public:
  Mole(int id_in)
    : id(id_in) {
    cout << "Mole ctor: "
         << id << endl;
  }

  ~Mole() {
    cout << "Mole dtor: "
         << id << endl;
  }

private:
  string id;
};

Mole * func() {
  Mole m(123);
  return new Mole(456);
}
```

```
int main() {
  Mole m(1);
  Mole *mPtr;
  // Line 1
  mPtr = func();
  // Line 2
  delete mPtr;
  // Line 3
  mPtr = new Mole(2);
  func();
  // Line 4
  delete mPtr;
  // Line 5
  cout << "all done!" << endl;
}
// Line 6 - after main returns
```

this is a bad behavior: we creat a new space in the heap for a mole object, but there are no pointer to ge the address of it' but for this question the number of mole object should +1.

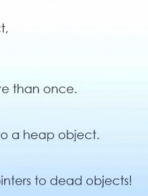
① local object 出 scope 自动 delete.

② dynamic object 手动 delete.

③ heap 中创建空间必须由外界接收地址.

Managing Dynamic Memory

- We're used to the compiler taking care of everything for us.
- Now, we have to make sure we clean up any dynamic memory when we're done with it.
- There are a few traps you might fall into...



Exercise: Whac-a-Mole

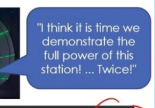
Slides

http://bit.ly/2MXQUGx

Question
Assume 8KB of stack space and 4MB of heap space. Assume an int is 4 bytes.

Which of these programs run out of memory and crash?

"I find your lack of memory disturbing..."



A
```
int main() {
  int *ptr;
  for (int i = 0; i < 1000000000; ++i) {
    ptr = new int(i);
  }
  delete ptr;
}
```
crash

B
```
void helper() {
  int *ptr = new int(10);
  ptr = new int(20);
  delete ptr;
}

int main() {
  for (int i = 0; i < 1000000000; ++i) {
    helper();
  }
}
```
crash

C
```
int main() {
  int x = 10000;
  for (int i = 0; i < 10000; ++i) {
    x = i;
  }
}
```

D
```
int main() {
  int arr[10000];
  for (int i = 0; i < 10000; ++i) {
    arr[i] = i;
  }
}
```
crash

E
```
int main() {
  int *arr = new int[10000];
  for (int i = 0; i < 10000; ++i) {
    arr[i] = i;
  }
  delete [] arr;
}
```
yucky

use "delete [ ] arr" to delete the arry in the heap space

① 如何在 heap 中定义 array

② 如何在 heap 中删除 array

Memory Leaks

- **Memory Leaks**
  Part of your code allocates dynamic memory, but neglects to free up the space when it's done.
- **Orphaned Memory**
  You lose the address of a heap object, meaning it is inevitably leaked.

This is getting out of hand. Now there are two of them!

The helper() function leaks memory. It allocates 2 ints, but only cleans up 1.

```
void helper() {
  int *ptr = new int(10);
  *ptr = new int(20);
  delete ptr;
}

int main() {
  for (int i = 0; i < 1000000000; ++i) {
    helper();
  }
}
```

When ptr switches from the 10 to the 20, the 10 is orphaned.

If leaky code is used frequently, your program can run out of memory!

5/25/2022

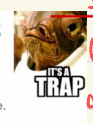
Orphaned memory lead to memory leaks.
But
Memory leak do not necessarily caused by the orphaned memory.

① Orphaned Memory 的定义.

② Orphaned Memory 和 memory leak 的关系.

Dynamic Memory Errors

- **Memory Leaks**
  Part of your code allocates dynamic memory, but neglects to free up the space when it's done.
- **Orphaned Memory**
  You lose the address of a heap object, meaning it is inevitably leaked.
- **Double Free**
  Attempt to delete a heap object more than once.
- **Bad Delete**
  Give delete an address not pointing to a heap object.
- **Dangling Pointers**
  Be careful not to dereference any pointers to dead objects!



① 5种有关 dynamic memory 的语法.

Fun Fact

- If you delete a null pointer (i.e. nullptr), nothing will happen.
  - A null pointer often represents "nothing there".
  - So delete ptr; means "delete the object there, if there is one".

this "delete" means if there is something in the dynamic space, just delete it, if is nothing there. do nothing

① delete a null pointer 的结果是什么.

Dangling Pointers

"I am altering the memory. Pray I don't alter it any further!"

- Deleting an object allows its memory to potentially be reused for other objects.
- As soon as an object's lifetime has ended, its data is no longer safe to access.
- However, pointers to that dead object may still be hanging around. We call these **dangling pointers**.

```
void example() {
  int x = 0;
  int *ptr = new int(42);
  delete ptr;
  int *ptr1 = new int(3);
  cout << *ptr << " "; // oops, we meant to type ptr1
  delete ptr1;
}
```

① dangling pointer 的定义

Exercise: Whac-a-Mole

Slides

http://bit.ly/3sZU9fV

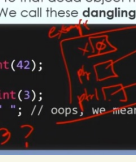
Question

Deleting an object twice usually results in a crash.

Deleting a non-heap object usually results in a crash.

Which of these program contain those runtime errors?

"I think it is time we demonstrate the full power of this station! ... Twice!"



A
```
int main() {
  int *ptr1 = new int(1);
  delete ptr1;
  ptr1 = new int(2);
  delete ptr1;
}
```

B
```
int main() {
  int *ptr1 = new int(1);
  delete ptr1;
  delete ptr1;
}
```

C
```
int main() {
  int x = 0;
  int *ptr1 = &x;
  delete ptr1;
}
```

D
```
int main() {
  int *ptr1 = new int(1);
  delete &ptr1;
}
```

E
```
int main() {
  int *ptr1 = new int(1);
  int *ptr2 = ptr1;
  delete ptr1;
  delete ptr2;
}
```

F
```
int main() {
  int *ptr;
  for (int i = 0; i < 10; ++i) {
    ptr = new int(i);
  }
  for (int i = 0; i < 10; ++i) {
    delete ptr;
  }
}
```
dangling last 10 lines

5/26/2022

36 Factory Pattern

- Dynamic memory is necessary when you don't know what you want until runtime.
- Example: **Factory Pattern**
  - Can create whatever subtype on the fly.
  - The object is passed between scopes.

```
Bird * Bird_factory(const string &color,
                    const string &name) {
  if (color == "blue") {
    return new BlueBird(name);
  }
  else if (color == "black") {
    return new Raven(name);
  }
}
```

```
int main() {
  Bird *birdPtr =
    Bird_factory("blue",
                 "Myrtle");

  // Use birdPtr

  delete birdPtr;
}
```

当 subtype polymophism 时，直到 runtime，我们才知道哪种 type. 所以 dynamic memory 在这种情况下重要

37 Dynamic Arrays

if you use the stack space, the size of the array must be fixed at the run time, which means, you can not use "cin" to control the size of the array in the running time.

- The size of a statically allocated array must be known at **compile time**.

```
int main() {
  const int NUM_ELEMENTS = 10;
  int arr[NUM_ELEMENTS];
}
```

cin

- The size of a dynamically allocated array may be determined at **runtime**.

```
int main() {
  cout << "How many elements? ";
  int howMany;
  cin >> howMany;
  int *arrPtr = new int[howMany];
}
```

①如何通过 dynamic memory 可以在 compile time 进行修改的特点，使 set 的 capacity 变得 flexible

38 Using Dynamic Arrays

- The result of a new expression that creates a dynamic array is a **pointer to the first element**.
- You can still use the [ ] operator to access elements!

```
int main() {
  cout << "How many elements? ";
  int howMany;
  cin >> howMany;
  int *arrPtr = new int[howMany];
  for (int i = 0; i < howMany; ++i) {
    arrPtr[i] = 42; // set each element to 42
  }
}
```

delete ?

5/26/2022

but we need to think about one question: how to delete this array

39 Dynamic Arrays and `delete[]`

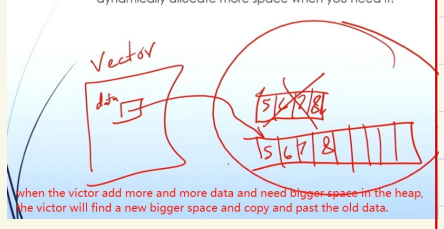
- Problem: The runtime environment is not able to distinguish between…
  - An `int*` pointing to a single `int` on the heap.
  - An `int*` pointing to the first element of a dynamic array on the heap.
- Solution: Use a special `delete[]` syntax for arrays.

```
int main() {
  ...
  int *ptr1 = new int(42);
  int *ptr2 = new int[howMany];

  delete ptr1; delete one single int element
  delete[] ptr2; delete the whole array
}
```

40 Growable Containers

- We'll see this next time – the basic idea is to dynamically allocate more space when you need it!



when the victor add more and more data and need bigger space in the heap, the victor will find a new bigger space and copy and past the old data.

5. Dynamic memory error↵
   ① Memory leak↵
   ② Orphaned memory ↵
   ③ Double delete ↵
   ④ Bad delete ↵
   ⑤ Dangling pointer: 一个 pointer 指向 heap 中的一个 object，通过 pointer delete 掉这个 object 之后，该 pointer 变成 dangling pointer，我们仍然可以用这个 pointer 指向其他 heap 中的 object，但是我们不能解引用已经被 delete 的地址上的 data。↵
   / delete nullptr: nothing will happen/↵
6. Factory pattern：当你在 compile time 时不知道具体数据类型，必须等到 run time 时，Dynamic memory 就很重要↵
7. Dynamic array:↵
   ① 不需要 fixed capacity：在 running time 时接收 capacity。↵
   ② 仍然可以使用[]来访问 array 中的 element↵
   ③ 初始化方法：int *arrPtr = new int[howMany]→从 heap 中传输出来的是 array 的第一个 element 的 address.↵
   ④ Delete：在删除时，为了和删除普通的 int 进行区分→ delete [] arrPtr;↵

Chapter 13 memory models and dynamic memory↵

1. 三种 storage duration↵
   ① Static(global)↵
   - Control by compiler ↵
   - Live for the whole program↵

   ② Automatic(local)↵
   - Control by the compiler ↵
   - Live inside a block (curly braces)↵
   - Live start initialization and dies when out of scope(for compound object, initialize: call constructor, when out of scope call: destructor) */the class defined at first will be deleted at last since the new class may depend on the old one/*↵

   ③ Dynamic ↵
   - Control by user↵

2. Memory address 的划分↵
   ① Stack↵
   - Automatic storage ↵
   - Grow dawn ↵

   ② The Void↵
   - Free space ↵

   ③ Heap↵
   - Dynamic storage ↵
   - Gown up ↵

   ④ Global ↵
   - Static storage ↵
   - Fixed size↵

   ⑤ Text↵
   - Machine code for your program ↵

3. Dynamic memory (heap)↵
   ① Create → new ↵
   Eg: int *ptr = new int (3);↵

↵
   ② Delete → delete↵
   Delete ptr;↵
   我们顺着 pointer 删除了 heap 地址上的 data，至于 pointer，我们可以让其等于 nullptr 或者让其 out of scope 被自动 delete。↵
   ③ 跟踪 heap 中数据的方法：we keep track of it using a pointer that hold its address.↵
↵
4. 造成 memory leak 的原因：↵
   ① Orphaned memory: you lose the address of the object in the heap.↵